
GT.M Bindings to Python

Release 1.0

Luis Ibanez
OSEHRA

January 1, 2013

Abstract

This paper describes an Open Source library to integrate the GT.M database with the Python language. The purpose of this integration is to facilitate the use of the GT.M database functionalities from Python scripts.

The source code is publicly available under the Apache 2.0 License, and it has been tested in Ubuntu Linux.

<https://github.com/OSEHR/m.js>

The bindings are implemented in C++, and are based on the C-Language API provided by GT.M. The Python interface is implemented by using SWIG wrapping of the C++ interface layer resulting in direct connection between GT.M and Python in the same process.

Contents

1	Introduction	2
2	GT.M C-API	2
3	SWIG Wrapping	3
4	Error Management	3
5	Configuration	4
6	How to Build	4
6.1	Download from Github	5
6.2	Install Dependencies	5
6.3	Setting the Environment	5
6.4	Configure and Build	5
6.5	A Quick Test	6
7	Examples	6
7.1	Example 1	7
7.2	Example 2	7

7.3	Example 3	7
7.4	Example 4	8
7.5	Example 5	8
7.6	Example 6	9
7.7	Example 7	10
7.8	Example 8	11
7.9	Example 9	11
7.10	Example 10	12
8	Conclusions	12

1 Introduction

M is both a Language and a Database. In this article, we are focused on the M Database, and in particular, on making possible to use the M Database from scripts written in the Python language.

This follows the pattern of usage of other popular NoSQL databases, such as MongoDB for example[2], that can be used from many different programming languages, event though there might be a preferred language to take full advantage of the database.

The integration is done here for the particular case of the GT.M implementation of the M Database.

2 GT.M C-API

The C-API of GT.M implemented by fisglobal is described in great detail in the following pages:

http://tinco.pair.com/bhaskar/gtm/doc/books/pg/UNIX_manual/ch11s05.html

The `m.js` library presented in this article is built upon the functionalities provided by this existing GT.M C-API. In particular, it takes advantage of the functions

- `gtm_init()`
- `gtm_cip()`
- `gtm_exit()`

and through them, it implements a C++ interface to the M functionalities of

- Get
- Set
- Lock

- Kill
- Order
- Query
- Execute

This API is implemented in a C++ class named “GTM” in the Source directory.

The connections to GT.M are defined in a file named `gtm_access.ci` also to be found in the Source directory.

This interface is the same that was described in a previous article [3].

3 SWIG Wrapping

The GTM C++ class described in the previous section, is wrapped here for Python using the popular Simplified Wrapper and Interface Generator (SWIG)[4].

Most of the configuration required to use SWIG with the C++ class is implemented via CMake scripts, and will be found in the `Wrapping` directory.

SWIG generates a shared library that Python can load as a module at run time. This library contains the interface that passes variables from Python to C++ and back, as methods of the C++ interface are called from Python.

Since the coding styles of Python and C++ are different, SWIG offers the possibility of translating the coding style during the wrapping process. In particular, it is customary in Python to name methods using lower case letters, instead of the typical camel-case style used in C++. These conversions are listed in the file `Python.i` in the `Wrapping` subdirectory.

Thanks to this conversion, the invocations in Python take the following form.

- `db.get`
- `db.set`
- `db.lock`
- `db.kill`
- `db.order`
- `db.query`
- `db.execute`

4 Error Management

Given that the integration is done as a C++ library, error management is implemented by taking advantage of Exceptions.

A C++ Exception class was customized by deriving from the `std::exception` class, and adding elements relevant to the GTM-Python integration. This class is called `GTMException` and can also be found in the Source directory.

The C++ exceptions are mapped to Python exceptions by SWIG, following the declarations in the file `Wrapping/gtm.i`.

```
// Use exceptions
#include <exception.i>

// Customize exception handling
%exception {
    try {
        $action
    } catch( std::exception &ex ) {
        char error_msg[1024];
        snprintf( error_msg, 1024, "Exception thrown in GTM $symname: %s", ex.what() );
        SWIG_exception( SWIG_RuntimeError, error_msg );
    } catch( ... ) {
        SWIG_exception( SWIG_UnknownError, "Unknown exception thrown in GTM $symname" );
    }
}
```

This code is described in detail in the following SWIG 2.0 documentation pages:

http://www.swig.org/Doc2.0/SWIGDocumentation.html#Customization_exception
http://www.swig.org/Doc2.0/SWIGDocumentation.html#Customization_nn7

5 Configuration

The configuration of the library is done with CMake[1].

In particular, this involved the creation of CMake descriptions on how to discover at configuration time the elements of the packages:

- GTM
- SWIG
- Python (development)

A good portion of this is done by the support that CMake has already built-in for SWIG.

6 How to Build

This build instructions are focused on Ubuntu Linux, which is the platform in which the library was developed and where it has been tested.

6.1 Download from Github

First you should download the source code from github with the command

```
git clone git://github.com/OSEHR/m.js.git
```

6.2 Install Dependencies

To prepare your build environment in Linux you should install the following packages:

- python-dev
- swig2.0

as well as installing GT.M.

6.3 Setting the Environment

The programs that use these bindings must be able to find at runtime the `gtm_access.ci` file.

To this end, GT.M uses the environment variable `GTMCi`, that must be set to the absolute path of the `gtm_access.ci` file.

For example as:

```
export GTMCi=/home/ibanez/bin/m.js/bin/gtm_access.ci
```

The other standard GT.M environment variables must be also set as usual. In particular

- `gtm_dist`
- `gtmgbldir`

6.4 Configure and Build

As with typical CMake configurations, one must start by creating a directory to host the binary build. This is referred to as the `BINARY` directory.

From that binary directory we can invoke `ccmake` and pass to it as argument the path to the directory where we put the source code of `m.js`. For example

```
ccmake /src/m.js
```

Then turn on the variable

```
WRAP_PYTHON
```

CMake should be able to find the Python development libraries and headers, as well as the SWIG 2.0 components. It might ask you for the location of `gtm_dist`.

You can provide this information, and then use the “c” key to configure and then the “g” key to generate Makefiles.

Once you return to the command line prompt, you can simply invoke “make” and the build should be complete in about 30 seconds.

The outcome of the build will be found in the `BINARY` directory, in the `lib` subdirectory. Additionally, an EGG installable version of the generated Python module will be found in the `Wrapping` subdirectory. This is useful for installing the module in a computer different from the one where it was built.

If you want to install the Python module in the same computer where you are building it, you can go to the binary directory of the build. Then enter the directory “Wrapping” and type the command

```
sudo python ./PythonPackage/setup.py install
```

This will install the files:

- `gtm.py`
- `gtm.pyc`

in the directory: `/usr/local/lib/python2.7/dist-packages`
and the file

- `_gtm.so`

in the directory: `/usr/lib/python2.7/dist-packages`

6.5 A Quick Test

The build process should generate in the `BINARY` directory a `lib` subdirectory where you will find among other things, the file:

- `runPythonTest.sh`

This file is a helper file to setup environment variables and to invoke a python script from the `Testing` directory.

7 Examples

This section presents several simple examples on how to interact with GT.M from Python using the bindings described in this article. You will find these examples in the `Testing` subdirectory of the `m.js` project.

7.1 Example 1

```
19
20 # load gtm module
21 from gtm import GTM
22
23 #
24 # Simply test constructor, destructor and connection to GT.M
25 #
26
27 db = GTM()
28
29 version = db.version()
30
31 print "Version = ", version
32
33 about = db.about()
34
35 print "About = ", about
```

7.2 Example 2

```
19
20 # load gtm module
21 from gtm import GTM
22
23 #
24 # Test the Set and Get methods
25 #
26
27 db = GTM()
28
29 db.set("^Capital","London")
30
31 capital = db.get("^Capital")
32
33 print "Capital = ", capital
```

7.3 Example 3

```
19
20 # load gtm module
21 from gtm import GTM
22
23 #
24 # Test the Set and Get methods
25 #
26
27 db = GTM()
28
29 globalName = "^Capital"
30 setValue = "London"
31
32 db.set( globalName, setValue )
```

```
33
34 getValue = db.get( globalName )
35
36 print globalName, " = ", getValue
```

7.4 Example 4

```
19
20 # load gtm module
21 from gtm import GTM
22
23 #
24 # Test the Set, Get and Kill methods
25 #
26
27 db = GTM()
28
29 globalName = "^Capital"
30 setValue = "London"
31
32 db.set( globalName, setValue )
33
34 getValue = db.get( globalName )
35
36 print globalName, " = ", getValue
37
38 db.kill( globalName )
```

7.5 Example 5

```
19
20 # load gtm module
21 from gtm import GTM
22
23 #
24 # Test the Set, Get and Kill methods
25 #
26
27 db = GTM()
28
29 #
30 # Exercise the string API
31 #
32
33 globalName = '^Capital("US")'
34 setValue = 'Washington'
35
36 db.set( globalName, setValue )
37
38 globalName = '^Capital("UK")'
39 setValue = 'London'
40
41 db.set( globalName, setValue )
42
```



```

43 getValue = db.order( globalName )
44
45 print "Order of ", globalName, " = ", getValue
46
47 expectedValue = "US"
48
49 db.kill('^Capital')
50
51 if getValue != expectedValue:
52     print "Test FAILED !"
53     print "Expected value = ", expectedValue
54     print "Received value = ", getValue
55     sys.exit(1)
56
57
58 #
59 #   Exercise the same pattern with direct strings
60 #
61
62 db.set( '^Capital("US")', 'Washington' )
63 db.set( '^Capital("UK")', 'London' )
64
65 getValue = db.order( '^Capital("UK")' )
66
67 print "Order of ", globalName, " = ", getValue
68
69 expectedValue = "US"
70
71 db.kill('^Capital')
72
73 if getValue != expectedValue:
74     print "Test FAILED !"
75     print "Expected value = ", expectedValue
76     print "Received value = ", getValue
77     sys.exit(1)

```

7.6 Example 6

```

19
20 # load gtm module
21 from gtm import GTM
22
23 #
24 #   Test the Set, Get and Query methods
25 #
26
27 db = GTM()
28
29 #
30 #   Exercise the string API
31 #
32
33 globalName = '^Capital("US")'
34 setValue = 'Washington'
35
36 db.set( globalName, setValue )
37

```

```

38 globalName = '^Capital("UK")'
39 setValue = 'London'
40
41 db.set( globalName, setValue )
42
43 getValue = db.query( globalName )
44
45 print "Query of ", globalName, " = ", getValue
46
47 expectedValue = '^Capital("US")';
48
49 db.kill('^Capital')
50
51 if getValue != expectedValue:
52     print "Test FAILED !"
53     print "Expected value = ", expectedValue
54     print "Received value = ", getValue
55     sys.exit(1)
56
57
58 #
59 #   Exercise the same pattern with direct strings
60 #
61
62 db.set( '^Capital("US")', 'Washington' )
63 db.set( '^Capital("UK")', 'London' )
64
65 getValue = db.query( '^Capital("UK")' )
66
67 print "Query of ", globalName, " = ", getValue
68
69 expectedValue = '^Capital("US")';
70
71 db.kill('^Capital')
72
73 if getValue != expectedValue:
74     print "Test FAILED !"
75     print "Expected value = ", expectedValue
76     print "Received value = ", getValue
77     sys.exit(1)

```

7.7 Example 7

```

19
20 # load gtm module
21 from gtm import GTM
22
23 #
24 #   Test the Execute method
25 #
26
27 db = GTM()
28
29 #
30 #   Exercise the string API
31 #
32

```

```
33 textOfCode = 'write $ZVERSION,!'  
34  
35 db.execute( textOfCode )  
36  
37 #  
38 #   Exercise the same pattern with direct strings  
39 #  
40  
41 db.execute( 'write $ZVERSION,!'
```

7.8 Example 8

```
19  
20 # load gtm module  
21 from gtm import GTM  
22  
23 #  
24 #   Test the Lock method  
25 #  
26  
27 db = GTM()  
28  
29 globalName = '^Capital("US")'  
30 setValue = 'Washington'  
31  
32 db.lock( globalName )  
33  
34 db.set( globalName, setValue )  
35  
36 db.kill( globalName )
```

7.9 Example 9

```
19  
20 # load gtm module  
21 from gtm import GTM  
22  
23 #  
24 #   Test the Lock method  
25 #  
26  
27 db = GTM()  
28  
29 globalName = "^ValueCounter"  
30 setValue = "0"  
31 getValue = "Initially empty"  
32  
33 db.lock( globalName )  
34  
35 db.set( globalName, setValue )  
36  
37 for i in xrange(0,9):  
38     db.execute("set ^ValueCounter=^ValueCounter+1")  
39     getValue = db.get( globalName )
```

```
40 print "counter = ", getValue
41
42 print "Final Counter Value = ", getValue
43
44 db.kill( "^ValueCounter" )
```

7.10 Example 10

```
19
20 # load gtm module
21 from gtm import GTM
22
23 db = GTM()
24
25 print db.about()
26
27 print db.version()
28
29 db.set("^FibonacciA", "1")
30 db.set("^FibonacciB", "1")
31
32 getValue = "Initially empty"
33
34 for i in xrange(1,10):
35     db.execute("set ^FibonacciValue=^FibonacciA+^FibonacciB")
36     db.execute("set ^FibonacciB=^FibonacciA")
37     db.execute("set ^FibonacciA=^FibonacciValue")
38     getValue = db.get("^FibonacciValue")
39     print "Fibonacci value = ", getValue
40
41 db.kill("^FibonacciA")
42 db.kill("^FibonacciB")
43 db.kill("^FibonacciValue")
```

8 Conclusions

The interface described in this article is based on a subset of operations defined in the `gtm_access.ci` file. These are so far very basic and low level operations. Given that there is a performance cost on going through the connection layers between Python and GTM, for actions that required higher performance, it might be desirable to customize those calls as one more entry in the `gtm_access.ci` table.

In general, this Python interface is offered with the goal of promoting the use of M in educational settings, in particular, to raise awareness about the value and importance of the M database.

Finally, given that the Python wrapping was implemented here using SWIG, we expect that it will be relatively easy to generate similar interfaces to the other many languages that SWIG support. In particular:

- Perl
- PHP
- Tcl

- Ruby
- Java
- Guile
- Mzscheme
- Chicken
- OCaml
- Pike
- C#

Please share your observations on how this interface could be improved.

References

- [1] CMake Multi-Platform Configuration. <http://www.cmake.org>, December 2012. 5
- [2] MongoDB Drivers. <http://www.mongodb.org/display/DOCS/Drivers>, December 2012. 1
- [3] GT.M Bindings to NodeJS. <http://hdl.handle.net/10909/54>, December 2012. 2
- [4] Simplified Wrapper and Interface Generator (SWIG). <http://www.swig.org>, December 2012. 3